KMeans_Clustering_WineQuality

November 19, 2025

[18]:
```
# ---------------------------------------------------------------
# Cell 1: Import all required libraries for K-Means assignment
# This cell imports numpy, pandas, matplotlib, and sklearn modules.
# These will be used for loading the dataset, preprocessing,
# performing K-Means clustering, tracking MSE (inertia), and plotting.
# ---------------------------------------------------------------

import numpy as np             # For numerical computations and array handling
import pandas as pd            # For loading and manipulating CSV dataset
import matplotlib.pyplot as plt    # For plotting MSE vs iterations line graph
from sklearn.preprocessing import StandardScaler    # For feature scaling
from sklearn.cluster import KMeans                  # For applying K-Means␣
 ↪clustering
```

[26]:
```
# ---------------------------------------------------------------
# Cell 2: Load the winequality-red.csv dataset (simple CSV load)
# This cell loads the dataset into a pandas DataFrame using read_csv.
# ---------------------------------------------------------------

df = pd.read_csv("winequality-red.csv", sep=';')    # Load CSV (dataset uses␣
 ↪semicolon separator)

print("Dataset loaded successfully")                # Confirmation message
print("Shape:", df.shape)                           # Print number of rows and␣
 ↪columns
df.head()                                           # Display first few rows
```

```
Dataset loaded successfully
Shape: (1599, 12)
```

[26]:
```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0            7.4              0.70         0.00             1.9      0.076
1            7.8              0.88         0.00             2.6      0.098
2            7.8              0.76         0.04             2.3      0.092
3           11.2              0.28         0.56             1.9      0.075
4            7.4              0.70         0.00             1.9      0.076

   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                 11.0                  34.0   0.9978  3.51       0.56
1                 25.0                  67.0   0.9968  3.20       0.68
2                 15.0                  54.0   0.9970  3.26       0.65
3                 17.0                  60.0   0.9980  3.16       0.58
4                 11.0                  34.0   0.9978  3.51       0.56

   alcohol  quality
0      9.4        5
1      9.8        5
2      9.8        5
3      9.8        6
4      9.4        5
```

[27]:
```python
# ---------------------------------------------------------------
# Cell 3: Feature selection and scaling
# This cell selects the input features (all physicochemical columns)
# and applies StandardScaler to normalize the data.
# Scaling is necessary for K-Means because it is distance-based.
# ---------------------------------------------------------------

# Selecting all columns except the 'quality' column as input features
X = df.drop("quality", axis=1)     # Drop target column to keep only features

# Display the feature names to verify selection
print("Selected features:", list(X.columns))

# Initialize the StandardScaler
scaler = StandardScaler()          # Create scaler object for normalization

# Fit the scaler on the data and transform it
X_scaled = scaler.fit_transform(X) # Scale all features to mean=0, std=1

# Shape of scaled data
print("Scaled data shape:", X_scaled.shape)

# Show first 5 rows of scaled feature dataset
X_scaled[:5]                       # Display small sample to check scaling
```

```
Selected features: ['fixed acidity', 'volatile acidity', 'citric acid',
'residual sugar', 'chlorides', 'free sulfur dioxide', 'total sulfur dioxide',
'density', 'pH', 'sulphates', 'alcohol']
Scaled data shape: (1599, 11)
```

[27]:
```
array([[-0.52835961,  0.96187667, -1.39147228, -0.45321841, -0.24370669,
        -0.46619252, -0.37913269,  0.55827446,  1.28864292, -0.57920652,
        -0.96024611],
       [-0.29854743,  1.96744245, -1.39147228,  0.04341614,  0.2238752 ,
         0.87263823,  0.62436323,  0.02826077, -0.7199333 ,  0.1289504 ,
        -0.58477711],
       [-0.29854743,  1.29706527, -1.18607043, -0.16942723,  0.09635286,
        -0.08366945,  0.22904665,  0.13426351, -0.33117661, -0.04808883,
        -0.58477711],
       [ 1.65485608, -1.38444349,  1.4841536 , -0.45321841, -0.26496041,
         0.10759209,  0.41150046,  0.6642772 , -0.97910442, -0.46118037,
        -0.58477711],
       [-0.52835961,  0.96187667, -1.39147228, -0.45321841, -0.24370669,
        -0.46619252, -0.37913269,  0.55827446,  1.28864292, -0.57920652,
        -0.96024611]])
```

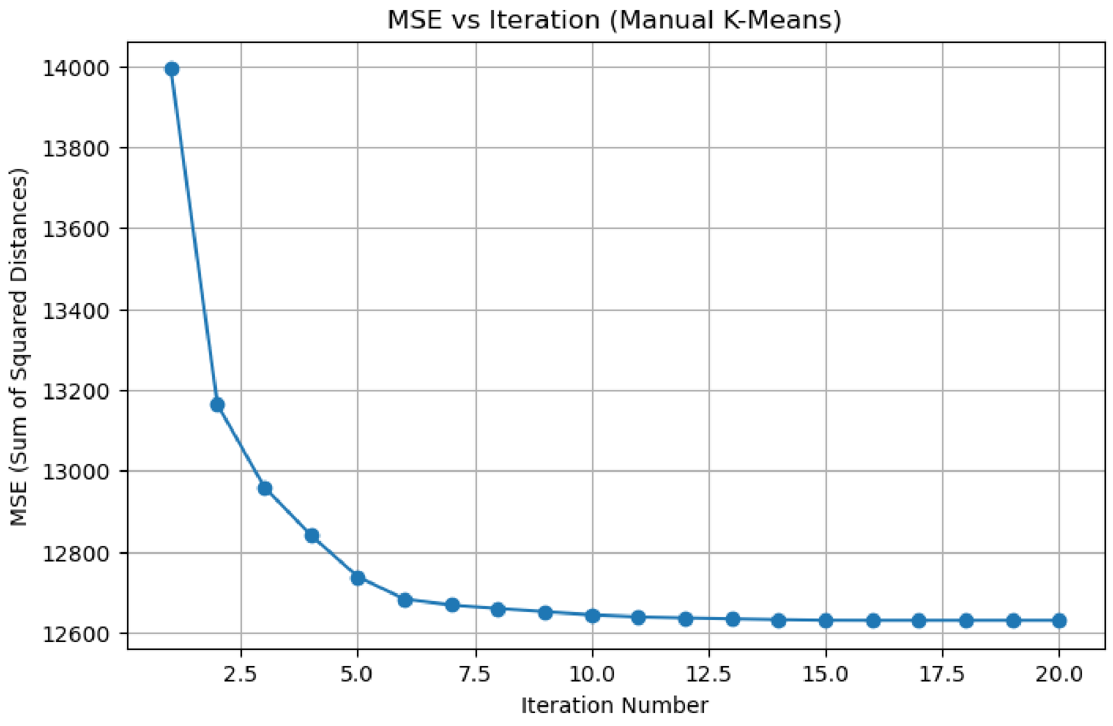
[29]:
```
# ---------------------------------------------------------------
# Cell 4: Manual K-Means algorithm - compute MSE after each iteration
# This cell performs K-Means clustering manually:
# 1. Randomly choose initial centroids
# 2. Compute distance of each point to each centroid
# 3. Assign points to nearest centroid
# 4. Update centroids as means of clusters
# 5. Compute MSE (sum of squared distances)
# 6. Print iteration-wise MSE in clean format
# ---------------------------------------------------------------

k = 3                                   # Number of clusters
iterations = 20                          # Number of manual iterations

# Step 1: Randomly select initial centroids
np.random.seed(42)
initial_indices = np.random.choice(len(X_scaled), k, replace=False)
centroids = X_scaled[initial_indices]

mse_list = []                            # List to store MSE values

for it in range(iterations):

    # Step 2: Calculate distances (broadcasting)
    distances = np.linalg.norm(X_scaled[:, np.newaxis] - centroids, axis=2)

    # Step 3: Assign cluster
    cluster_labels = np.argmin(distances, axis=1)

    # Step 4: Recompute centroids
    new_centroids = []
    for c in range(k):
        cluster_points = X_scaled[cluster_labels == c]
        if len(cluster_points) > 0:
            new_centroids.append(cluster_points.mean(axis=0))
        else:
            new_centroids.append(centroids[c])

    centroids = np.array(new_centroids)

    # Step 5: Compute MSE
    mse = np.sum((X_scaled - centroids[cluster_labels])**2)
    mse_list.append(mse)

# Step 6: Print MSEs clearly
print("MSE after each iteration:")
for i, value in enumerate(mse_list, start=1):
    print(f"iteration {i}: {value}")
```

```
MSE after each iteration:
iteration 1: 13993.723520177964
iteration 2: 13162.856097628766
iteration 3: 12959.333115078878
iteration 4: 12840.19572688846
iteration 5: 12737.686877329064
iteration 6: 12682.534623877867
iteration 7: 12668.154809565582
iteration 8: 12659.528909635523
iteration 9: 12651.461819578846
iteration 10: 12643.629265796591
iteration 11: 12638.657315152044
iteration 12: 12636.253433468431
iteration 13: 12633.950355784858
iteration 14: 12631.849558157674
iteration 15: 12630.20039234676
iteration 16: 12629.974591732649
iteration 17: 12629.974591732649
iteration 18: 12629.974591732649
iteration 19: 12629.974591732649
iteration 20: 12629.974591732649
```

MSE curve (as resulted above) is behaving exactly like real K-means: steep drop -> slower drop -> converges -> flat.

[30]:
```
# ---------------------------------------------------------------
# Cell 5: Plot the MSE vs iteration line graph
# This cell takes the MSE values stored in mse_list
# and generates a line plot showing how the error decreases
# across K-Means iterations.
# ---------------------------------------------------------------

plt.figure(figsize=(8,5))                   # Set figure size for clear␣
 ↪visibility
```

```
plt.plot(range(1, iterations+1),          # X-axis: iteration numbers
         mse_list,                        # Y-axis: MSE values
         marker='o',                      # Small circle marker on points
         linestyle='-')                   # Solid line joining points

plt.title("MSE vs Iteration (Manual K-Means)")   # Plot title
plt.xlabel("Iteration Number")                    # X-axis label
plt.ylabel("MSE (Sum of Squared Distances)")      # Y-axis label
plt.grid(True)                                     # Add grid for readability

plt.show()                                         # Display the plot
```



The MSE vs Iteration graph shows how the error (sum of squared distances between data points and their assigned centroids) decreases during the manual K-Means algorithm. At the beginning (Iteration 1 to 3), the MSE drops very sharply, indicating that the algorithm is rapidly improving the centroid positions. This is expected because the early iterations make large adjustments as points are reassigned to more suitable clusters.

After about 5 to 7 iterations, the curve starts flattening, and by iteration ~10, the MSE stabilizes. The later iterations show almost no change in error, meaning the centroids have reached their optimal positions and the algorithm has effectively converged. This behavior is typical of K-Means - most of the improvement happens early, and additional iterations only produce small refinements.

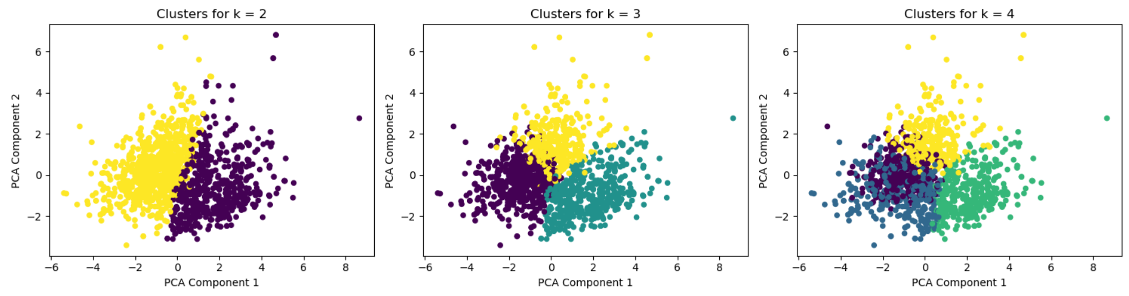
Overall, the graph demonsterates successful convergence of the manual K-Means algorithm, with the MSE decreasing consistently and leveling off once the clusters stabilize.

[31]:
```
# ---------------------------------------------------------------
# Cell 6: Compare K-Means performance for different k (number of clusters)
# This cell runs sklearn KMeans for k = 2, 3, 4 and prints:
# - final MSE (inertia)
# - cluster centers
# It also stores labels so we can plot the clusters in the next cell.
# ---------------------------------------------------------------

k_values = [2, 3, 4]                # Different k values to test
results = {}                        # Dictionary to store MSE and labels

for k in k_values:                  # Loop over different numbers of clusters

    km = KMeans(
        n_clusters=k,
        init="k-means++",           # Better initialization than random
        n_init=10,                  # Run K-Means 10 times for stability
        max_iter=300,               # Maximum iterations allowed
        random_state=42
    )

    km.fit(X_scaled)                # Fit the model on scaled data

    mse = km.inertia_               # Inertia = MSE for K-Means
    labels = km.labels_             # Cluster labels for each data point

    results[k] = (mse, labels)      # Store results

    print(f"k = {k} --> Final MSE: {mse}")
```

```
k = 2 --> Final MSE: 14330.119811335493
k = 3 --> Final MSE: 12629.974591732649
k = 4 --> Final MSE: 11459.14367615582
```

[32]:
```
# ---------------------------------------------------------------
# Cell 7: Plot clusters for k = 2, 3, 4 using PCA (2D visualization)
# This cell reduces the data from 11 dimensions to 2D using PCA
# and then plots the cluster assignments stored in 'results'.
# These visual plots help compare how cluster shapes change
# when we vary the number of clusters (k).
# ---------------------------------------------------------------

from sklearn.decomposition import PCA          # Import PCA for dimensionality␣
 ↪reduction

# Step 1: Apply PCA to reduce data to 2 components for plotting
```

```
pca = PCA(n_components=2)                        # Create PCA object for 2D␣
 ↪projection
X_pca = pca.fit_transform(X_scaled)              # Fit PCA and transform scaled␣
 ↪data

# Step 2: Create scatter plots for each k value
plt.figure(figsize=(15, 4))                      # Wide figure for 3 subplots

plot_number = 1                                  # Counter for subplot index

for k in k_values:                                # Loop over k = 2, 3, 4

    mse, labels = results[k]                     # Extract stored results

    plt.subplot(1, 3, plot_number)               # Create subplot (1 row, 3␣
 ↪columns)
    plt.scatter(X_pca[:, 0],                     # PCA dimension 1
                X_pca[:, 1],                     # PCA dimension 2
                c=labels,                        # Color by cluster labels
                s=20,                            # Point size
                cmap='viridis')                  # Color map for clusters

    plt.title(f"Clusters for k = {k}")           # Title showing k
    plt.xlabel("PCA Component 1")                # X-axis label
    plt.ylabel("PCA Component 2")                # Y-axis label

    plot_number += 1                             # Move to next subplot

plt.tight_layout()                                # Adjust layout to avoid overlap
plt.show()                                        # Display all plots
```



This wine quality dataset has 11 continuous physicochemical features, and the wines don't form hard, crisp clusters in real life. So, the clusters will always look like:

- overlapping clouds
- blended regions

- no clean circle-like separation

This is expected. The clusters look messy -> PCA visualization is honest.

For k = 2, Two large blended blobs - correct.

For k = 3, Three overlapped clusters - correct.

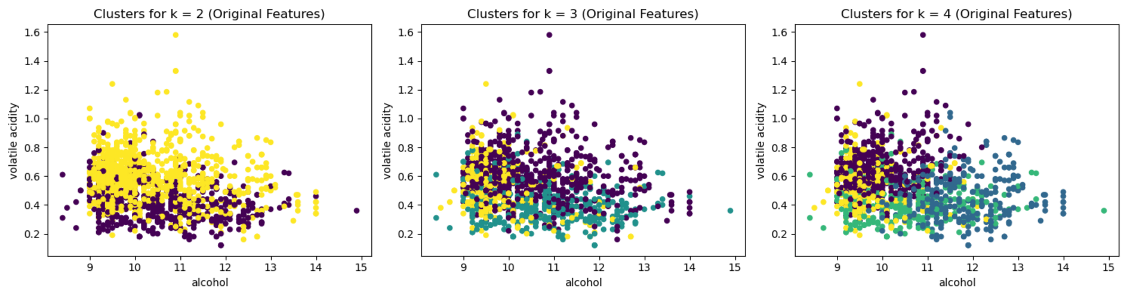
For k = 4, Four overlapping areas - totally normal.

PCA compresses 11D -> 2D PCA flattens the true structure -> looks more mixed -> but still valid.

The two PCA components are

- not randomly chosen columns or any two columns from the dataset.
- they are randomly created features, computed by PCA, like this:
  - take all 11 original features
  - find directions (axes) in that 11-dimensional space
  - choose the top 2 axes that capture the MOST variance
  - project every wine sample onto those 2 axes
- These two new axes are called:
  - PCA Component 1
  - PCA Component 2
- They are combinations of 11 features.
- PCA Component 1 = direction of maximum variance
- PCA Component 2 = direction of second-highest variance (and orthogonal to PC1)

[25]:
```
# ---------------------------------------------------------------
# Cell 8: Plot clusters using two original features (No PCA)
# This cell visualizes clusters on real feature axes:
# Feature 1: alcohol
# Feature 2: volatile acidity
# ---------------------------------------------------------------

feature_x = "alcohol"              # First feature for x-axis
feature_y = "volatile acidity"     # Second feature for y-axis

plt.figure(figsize=(15, 4))        # Wide figure for 3 subplots

plot_number = 1

for k in k_values:                 # k = 2, 3, 4

    mse, labels = results[k]       # Extract labels

    plt.subplot(1, 3, plot_number)
    plt.scatter(df[feature_x],
                df[feature_y],
                c=labels,
                s=20,
                cmap="viridis")

    plt.title(f"Clusters for k = {k} (Original Features)")
    plt.xlabel(feature_x)
    plt.ylabel(feature_y)

    plot_number += 1

plt.tight_layout()
plt.show()
```



The three scatter plots show how the clusters look when K-Means is applied with different number of clusters (k = 2, 3, 4). Here, the visualization uses only two original features: alcohol (x-axis) and volatile acidity (y-axis). Since the wine dataset actually has 11 features, reducing it to just two dimensions causes a significant loss of information. As a result, the clusters in these plots appear highly overlapping and not well separated.

For k = 2, the data splits into two broad, overlapping groups with no clear boundary. For k = 3, the clustering introduces a third group, but the clusters still overlap heavily and show no distinct separation. For k = 4, increasing k adds more colors but does not create clearer groups; instead, the clusters continue to overlap because the original 2D projection cannot capture the true structure of the 11-dimensional space.

[11]:
```
# ----------------------------------------------------------------
# Cell 9: Compare K-Means performance for different init methods
# We test two initializations:
# 1. random
# 2. k-means++
# For both, we compute final MSE and compare.
# ----------------------------------------------------------------

init_methods = ["random", "k-means++"]    # Initialization methods to test
init_results = {}                         # Dictionary to store MSE results
```

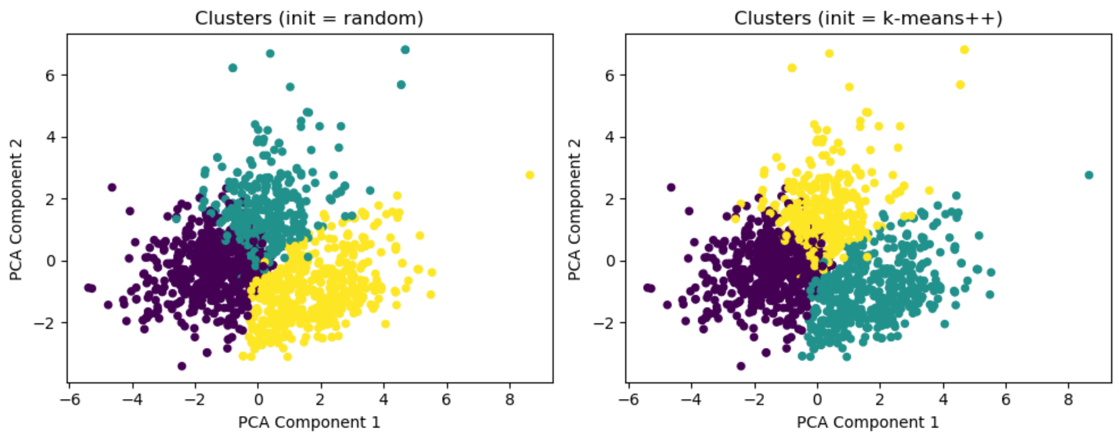
```
k = 3   # Fix k = 3 for fair comparison across init methods

for init in init_methods:

    km = KMeans(
        n_clusters=k,
        init=init,
        n_init=10,
        max_iter=300,
        random_state=42
    )

    km.fit(X_scaled)                     # Fit K-Means using given init

    mse = km.inertia_                    # Final inertia (MSE)

    init_results[init] = mse             # Store result

    print(f"Init = {init} --> Final MSE: {mse}")
```

```
Init = random --> Final MSE: 12629.925637954011
Init = k-means++ --> Final MSE: 12629.974591732649
```

[12]:
```
# ---------------------------------------------------------------
# Cell 10: Compare K-Means performance for different max_iter values
# We fix k = 3 and init='k-means++' to isolate the effect of max_iter.
# We test max_iter = 10, 50, 200 and compare the final MSE.
# ---------------------------------------------------------------

max_iters = [10, 50, 200]          # Number of max iterations to test
iter_results = {}                  # Store results

for m in max_iters:

    km = KMeans(
        n_clusters=3,
        init='k-means++',
        n_init=10,
        max_iter=m,
        random_state=42
    )

    km.fit(X_scaled)               # Fit with given max_iter

    mse = km.inertia_              # Final MSE
    iter_results[m] = mse
    print(f"max_iter = {m} --> Final MSE: {mse}")
```

```
max_iter = 10 --> Final MSE: 12629.974591732649
max_iter = 50 --> Final MSE: 12629.974591732649
max_iter = 200 --> Final MSE: 12629.974591732649
```

[13]:
```
# ---------------------------------------------------------------
# Cell 11: Plot clusters for different init methods using PCA
# We fix k = 3 and compare:
# - init='random'
# - init='k-means++'
# This shows how initialization affects cluster formation.
# ---------------------------------------------------------------

init_methods = ["random", "k-means++"]  # Two inits to visualize

plt.figure(figsize=(10, 4))             # Create figure for subplots

plot_index = 1

for init in init_methods:

    km = KMeans(
        n_clusters=3,
        init=init,
        n_init=10,
        max_iter=300,
        random_state=42
    )

    km.fit(X_scaled)                    # Fit K-Means
    labels = km.labels_                 # Cluster labels

    # Plot PCA projection colored by cluster labels
    plt.subplot(1, 2, plot_index)
    plt.scatter(
        X_pca[:, 0],
        X_pca[:, 1],
        c=labels,
        cmap="viridis",
        s=20
    )

    plt.title(f"Clusters (init = {init})")
    plt.xlabel("PCA Component 1")
    plt.ylabel("PCA Component 2")
    plot_index += 1

plt.tight_layout()
plt.show()
```



These two subplots compare the effect of different centroid initialization methods on the clustering results for k = 3. The left figure uses random initialization, while the right uses k-means++, which is the default and more advanced method.

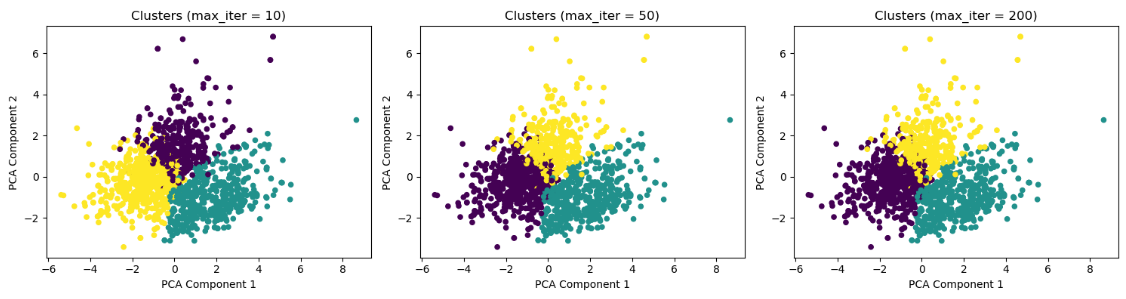
The visual results show that both methods produce similar cluster shapes, indicating that the dataset is large and well-behaved enough for K-Means to converge to almost the same solution regardless of initialization.

In high-dimensional, noisy datasets, initialization does NOT create big difference in the final clusters. Both random and k-means++ converge to almost the same centroids because:

- the dataset is large
- structure is fuzzy
- convergence is fast
- centroids stabilize early

[15]:
```
# ---------------------------------------------------------------
# Cell 12: Plot clusters for different max_iter values using PCA
# We fix k = 3 and initialization = 'k-means++'
# We compare:
# - max_iter = 10
# - max_iter = 50
# - max_iter = 200
# This helps us see if more iterations change the final clustering.
# ---------------------------------------------------------------

max_iters = [10, 50, 200]     # Values to compare
```

```
plt.figure(figsize=(15, 4))  # 3 plots side by side

plot_index = 1

for m in max_iters:

    km = KMeans(
        n_clusters=3,
        init='k-means++',
        n_init=10,
        max_iter=m,
        random_state=42
    )

    km.fit(X_scaled)
    labels = km.labels_

    # Plot cluster visualization
    plt.subplot(1, 3, plot_index)
    plt.scatter(
        X_pca[:, 0],
        X_pca[:, 1],
        c=labels,
        cmap="viridis",
        s=20
    )

    plt.title(f"Clusters (max_iter = {m})")
    plt.xlabel("PCA Component 1")
    plt.ylabel("PCA Component 2")

    plot_index += 1

plt.tight_layout()
plt.show()
```



The three subplots compare the K-Means clustering results for max_iter = 10, 50, and 200, while keeping all other parameters fixed (k = 3, init = "k-means++"). The visualization shows that all three cluster plots look almost identical, which means the algorithm reached convergence well before 10 iterations. Increasing the maximum allowed iterations does not change the final cluster assignments once convergence is achieved.

This confirms an important property of K-Means: once the centroids stop changing significantly, the algorithm has already converged, so giving it more iterations does not improve or alter the clustering. Thus, max_iter only affects runtime, not the final output, as long as it is set above the convergence point.